Solving for **Appendix A – Java Code**

Code for Trajectory Using Law of Sines and Law of Cosines and Pi Space Examples

Note: This is an old snapshot of the code. The latest code can be viewed and downloaded at https://github.com/pispacesw

```
package pispace.core;

import java.util.ArrayList;
import java.util.HashMap;


import java.math.BigDecimal;
import java.math.BigInteger;
import java.math.MathContext;

import org.apache.log4j.Logger;

import pispace.core.trajectory.PiSpaceCriteria;
import pispace.core.trajectory.PiSpaceObjectTrajectoryInfo;


/*
 *
 *
 * Core class for the Pi-Space Theory.
 *
 * @author Martin Brady
 *
 *
 * This class is a core Pi-Space class with implementations of its
 * important functions.  The Pi-Space versions
 * are derived in the Pi-Space Physics Theory in the Advanced
 * Formulas section which contains the complete list.
 *
 * Currently, this class is a work in progress and I will add
 * the functions over time.
 *
 *
 * TERMS AND CONDITIONS: PLEASE READ
 *
 *  Although this code may be freely available to use for test applications,
 *  this code is NOT free to use in commercial applications and will require
 *  payment to the author.  The code is based on the Pi-Space Physics Theory
 *  which is (Copyright) Martin Brady.
```

```
 *
 */

public class PiSpaceFormulas {

	static Logger logger = Logger.getLogger(pispace.core.PiSpaceFormulas.class);

	public final double SPEED_OF_LIGHT_KILOMETRES_PER_SECOND = 300000.0;
	public final double SPEED_OF_LIGHT_METRES_PER_SECOND = 299792458.0;
	public final double SPEED_OF_LIGHT_MILES_PER_SECOND = 186000.0;
	public final double SPEED_OF_LIGHT_FEET_PER_SECOND = 983571056.0;

	public static final int UNIT_METRES_PER_HOUR = 1;
	public static final int UNIT_MILES_PER_HOUR = 2;
	public static final int UNIT_FEET_PER_HOUR = 3;
	public static final int UNIT_FEET_PER_SECOND = 4;

	public static final double UGC_MILES =  3.439E-11;
	public static final double UGC_METRES = 6.67300E-11;

	public int unit;

	/*
	 *
	 * constructor
	 *
	 */
	public PiSpaceFormulas(int unit) {

		setUnit(unit);

	}


	public static Logger getLogger() {
		return logger;
	}

	public static void setLogger(Logger logger) {
		PiSpaceFormulas.logger = logger;
	}

	/*
	 *
	 * Metres or Miles?
	 *
```

```
 */
public void setUnit(int unit) {
	this.unit = unit;
}

public int getUnit() {
	return unit;
}

/*
 *
 * Hours to seconds
 *
 */
public double getHoursToSeconds(double hours) {
	// return hours;
	return hours / 3600.0;
}


/*
 *
 * Seconds to hours
 *
 */
public double getSecondsToHours(double seconds) {
	// return seconds;
	return seconds * 3600.0;
}

/*
 *
 * Diameter to energy form
 *
 */
public double getCSquared() {
	if (getUnit() == UNIT_METRES_PER_HOUR) {
		return SPEED_OF_LIGHT_METRES_PER_SECOND
				* SPEED_OF_LIGHT_METRES_PER_SECOND;
	} else if (getUnit() == UNIT_FEET_PER_SECOND) {
		return SPEED_OF_LIGHT_FEET_PER_SECOND
				* SPEED_OF_LIGHT_FEET_PER_SECOND;
	} else {
		return SPEED_OF_LIGHT_MILES_PER_SECOND
				* SPEED_OF_LIGHT_MILES_PER_SECOND;
	}
```

```
}

/*
 *
 * Speed of Light in chosen units
 *
 */
public double getC() {
	if (getUnit() == UNIT_METRES_PER_HOUR) {
		return SPEED_OF_LIGHT_METRES_PER_SECOND;
	} else if (getUnit() == UNIT_FEET_PER_SECOND) {
		return SPEED_OF_LIGHT_FEET_PER_SECOND;
	} else {
		return SPEED_OF_LIGHT_MILES_PER_SECOND;
	}
}

/*
 *
 * Map from Pi-Space energy to traditional Newtonian energy
 *
 */
public double getPiSpaceEnergyToNewton(double energy) {
	return energy * getCSquared();
}

/*
 *
 * Convert a Velocity into an energy format
 *
 */
public double getPiSpaceVelocityToEnergy(double velocityPercentLight) {
	return ((velocityPercentLight * velocityPercentLight) / (getCSquared()));
}
```

.....

```
/*
 *
```

```
	* Pass in a PiSpaceObjectTrajectoryInfo object and this method will calculate the next
	* position by returning a modified copy of the object.
	*
	* position x1, y1 cog x1,y1 velocity of object kmps angle movement wrt to
	* gravity field acceleration KPMS (e.g. Gravity Field)
	*
	* This algorithm is based on the Orbit document in the Pi-Space Physics
	* Theory
	*
	* Note: The smaller the value of time t, the more accurate the trajectory.
	*
	*/
	public PiSpaceObjectTrajectoryInfo getNewtonianPiSpaceTrajectoryNextPosition(
				PiSpaceObjectTrajectoryInfo vals
				) {

			getLogger()
					.debug("---------------------------------------------------------------- ITERATION START
");

			PiSpaceObjectTrajectoryInfo h = new PiSpaceObjectTrajectoryInfo();

			double time = vals.getMilliseconds() / 1000.0;

			// Calculate COG distance using COG x1,y1 and satellite x1,y1

			double distanceFromCOG =
					getDistanceBetweenTwoPoints(
							vals.getX1(),
							vals.getY1(),
							vals.getX1COG(),
							vals.getY1COG());

			//
			// store values for next iteration
			//

			h.setAngleWRTGravity(vals.getAngleWRTGravity());

			h.setMilliseconds(vals.getMilliseconds());

			h.setX1COG(vals.getX1COG());

			h.setY1COG(vals.getY1COG());

			h.setRadius(vals.getRadius());
```

```
h.setMass1(vals.getMass1());

h.setMass2(vals.getMass2());

h.setCriterias(vals.getCriterias());

h.setCurrentVelocity(vals.getCurrentVelocity());

h.setInitialAngleWRTGravity(vals.getInitialAngleWRTGravity());

getLogger().debug("Object initial velocity is " + vals.getInitialObjectVelocity());
getLogger().debug("Milliseconds is " + vals.getMilliseconds());
getLogger().debug("Time milliseconds/1000 is " + time);
getLogger().debug("AngleAxesOffset from field is " + vals.getAngleAxesOffset());
getLogger().debug("DistanceFromCOG is " + distanceFromCOG);
getLogger().debug("Radius of planet is " + vals.getRadius());
getLogger().debug("AngleWRTGravityField is " + vals.getAngleWRTGravity());
getLogger().debug("Initial AngleWRTGravityField is " + vals.getInitialAngleWRTGravity());
getLogger().debug("Mass of planet is " + vals.getMass2());
getLogger().debug("Criterias are " + vals.getCriterias().size());
getLogger().debug("Current Field Velocity is " + vals.getCurrentVelocity());
getLogger().debug("");



//
// velocity based distance for field movement
//

double vt =  0;

double fieldVelocity =
                        (vals.getInitialObjectVelocity()
                        *
                        Math.cos(convertDegreesToRadians(vals.getInitialAngleWRTGravity()))*(-1.0));

h.setInitialConstantFieldVelocity(fieldVelocity);
getLogger().debug("Initial constant field velocity is " + h.getInitialConstantFieldVelocity());

if (vals.isUsePiSpaceFormulas()) {
        vt = this.getPiSpaceDistanceUseNewtonianUnits(fieldVelocity, 0, time, true);
} else {
        vt = this.getNewtonianDistance(fieldVelocity, 0, time);
}

getLogger().debug("V*T field distance is " + vt);
```

```
getLogger().debug("");


//
// velocity based distance for particle movement, does not change
//

double particlevt =  0;

double particleVelocity =
                        (vals.getInitialObjectVelocity()
                        *
                        Math.sin(convertDegreesToRadians(vals.getInitialAngleWRTGravity())));

h.setInitialConstantParticleVelocity(particleVelocity);
getLogger().debug("Initial constant particle velocity is " + h.getInitialConstantParticleVelocity());

if (vals.isUsePiSpaceFormulas()) {
            particlevt = this.getPiSpaceDistanceUseNewtonianUnits(particleVelocity, 0, time, true);
} else {
            particlevt = this.getNewtonianDistance(particleVelocity, 0, time);
}

getLogger().debug("V*T particle distance is " + particlevt);
getLogger().debug("");

//
// determine gravity at this point
//

double grav = getNewtonianGravity(vals.getMass1(), vals.getMass2(), distanceFromCOG);
getLogger().debug("Derived gravity for this position is "+grav);


//Y velocity relative to gravity field e.g. straight up
//We need to keep track of this from the previous moment

double velocityInYDirection = this.getNewtonianFinalVelocity(
                        vals.getFieldVelocity(), grav, vals.getFieldDistance());

getLogger().debug(
                        "Field velocity for next step is "+
                        velocityInYDirection+
                        " was "+
                        vals.getFieldVelocity()+
                        " for distance "+vals.getFieldDistance());
```

```
h.setFieldVelocity(velocityInYDirection);

//
// just focus on distance due to field acceleration and time
//

double accDist = 0;

if (vals.isUsePiSpaceFormulas()) {
        accDist = this.getPiSpaceDistanceUseNewtonianUnits(velocityInYDirection, grav, time, true);
} else {
        accDist = this.getNewtonianDistance(velocityInYDirection, grav, time);
}

h.setFieldDistance(accDist);

getLogger().debug("Acceleration field distance is " + accDist);



//
// path of least time...
// do we continue to move in the direction we are travelling in?  or does gravity take over?
//
getLogger().debug("");
double pathOfLeastTimeAngle = this.getPiSpacePathOfLeastTime(
                vt, accDist, vals.getAngleWRTGravity());

vals.setAngleWRTGravity(pathOfLeastTimeAngle);

// ****
// **** Step 1: Calculate distance u and new angle WRT gravity Alpha
// ****

getLogger().debug("");
getLogger()
                .debug("Step 1: Calculate distance u and new angle WRT gravity Alpha");

// First calculate u = new velocity per second

// Rule of thumb: Extract the field first, then apply the angle WRT to
// gravity, produces 180-angleWRTGravity
// Reason to do it this way, we get acceleration in the direction of
// movement towards COG
// while using the Law of the Cosines
```

```
double interiorAngle = 180 - vals.getAngleWRTGravity();

getLogger().debug("Interior angle is "+interiorAngle);
getLogger().debug("");

double fieldDistance = vt - accDist;

double u = getLawOfCosinesDistance(particlevt, fieldDistance, interiorAngle);
getLogger().debug(
                        "Distance u which is distance between particle lengths " + particlevt
                        + " and field length " + (fieldDistance) + " having interior angle " + interiorAngle
                        + " is " + u);
getLogger().debug("u is "+u);
getLogger().debug("");

//for now only consider y velocity


double thevelocityInYDirection = this.getNewtonianFinalVelocity(
                        h.getFieldVelocity(), grav, h.getFieldDistance());

getLogger().debug("The current opposing field velocity is "+thevelocityInYDirection);

h.setCurrentVelocity(h.getInitialConstantFieldVelocity()-thevelocityInYDirection);
getLogger().debug("Current field velocity is (part which changes) "+h.getCurrentVelocity());
getLogger().debug("");


//
//Calculate the Alpha angle
//
double pureAlphaAngle = getLawOfSinesAngle(u, interiorAngle, particlevt);
getLogger().debug("Alpha angle is "+pureAlphaAngle);
getLogger().debug("");

//initial velocity stays constant
h.setInitialObjectVelocity(vals.getInitialObjectVelocity());

// Next calculate the new angleWRTGravity called alpha
double betaAngle = getLawOfSinesAngle(u, interiorAngle, fieldDistance);
getLogger().debug("Beta angle is "+betaAngle);
getLogger().debug("");

// Use accDist, it's better to use than v*t...
double alphaFromBeta = 180 - betaAngle - interiorAngle;
```

```
getLogger().debug("Setting new angleWRTGravity (oldAngle|newAngle) "
                + vals.getAngleWRTGravity() + "/" + alphaFromBeta);

// new angleWRTGravity
h.setAngleWRTGravity(alphaFromBeta);

// ****
// **** Step 2: Calculate length r which is the distance to the COG,
// from this calculate the new length s which is the new distance to COG
// ****

getLogger().debug("");
getLogger()
                .debug("Step 2: Calculate length r which is the distance to the COG, from this
calculate the new length s which is the new distance to COG");

double r = distanceFromCOG;

//
// Simple collision detection
//

if (accDist > r) {
        getLogger()
                        .debug("!!!!Gravity pulled object through the COG origin!!!! Collision
detection.");
        // Right now this is how we can end
        h.setCollisionDetection(true);

} else {
        h.setCollisionDetection(false);
}

//
// new distances
//

double s = getLawOfCosinesDistance(u, r, alphaFromBeta);
h.setDistanceFromCOG(s);
getLogger().debug("(r(old distanceFromCOG)|s(new distanceFromCOG)) "
                + r + "/" + s);

// ****
// **** Step 3: Calculate new angle WRT to axes, new theta
// ****
```

```
getLogger().debug("");
getLogger().debug("Step 3: Calculate new angle WRT to (field) axes, new theta");

double newAngleWRTAxes = getLawOfSinesAngle(s, alphaFromBeta, u);

getLogger().debug("(angleWRTFieldAxes|newAngleWRTFieldAxes) " + vals.getAngleAxesOffset()
                        + "/" + newAngleWRTAxes);

double totalOffset = (vals.getAngleAxesOffset() + newAngleWRTAxes) % 360;
getLogger().debug("Total axis offset due to particle movement " + totalOffset);
getLogger().debug("");

h.setAngleAxesOffset(totalOffset);

// final step 4
// calculate the new x,y position of the object moving the
// gravity field
// we have an angle and a length (polar co-ordindates) so we can covert
// into
// a new x,y position.

double rAngle = convertDegreesToRadians(90 - totalOffset);

getLogger().debug("X (s|90-totalOffset|Math.cos) " + s + "/"
                        + (90 - totalOffset) + "/" + Math.cos(rAngle));

getLogger().debug("Y (s|90-totalOffset|Math.sin) " + s + "/"
                        + (90 - totalOffset) + "/" + Math.sin(rAngle));
getLogger().debug("");

double newx = s * Math.cos(rAngle) + vals.getX1COG();
double newy = s * Math.sin(rAngle) + vals.getY1COG();

getLogger().debug("newx relative to fixed axis " + newx);
getLogger().debug("newy relative to fixed axis " + newy);
getLogger().debug("");
h.setX1(newx);
h.setY1(newy);

double deltax = newx - vals.getX1();
double deltay = newy - vals.getY1();

getLogger().debug("deltax " + deltax);
getLogger().debug("deltay " + deltay);
getLogger().debug("");
```

```
h.setDeltaY(deltax);
h.setDeltay(deltay);

h.setTotalDeltaX(h.getDeltaX()+vals.getTotalDeltaX());
getLogger().debug("Total Delta X is " + h.getTotalDeltaX());
h.setTotalDeltaY(h.getDeltay()+vals.getTotalDeltaY());
getLogger().debug("Total Delta Y is " + h.getTotalDeltaY());
getLogger().debug("");

double rememberTime = vals.getTotalTime();
h.setTotalTime(vals.getTotalTime()+time);
getLogger().debug("Total time is "+h.getTotalTime()+" ("+rememberTime+"+"+time+")");
getLogger().debug("");


h.setDeltaDistance(u);
getLogger().debug("Delta distance from COG is "+h.getDeltaDistance());

getLogger().debug("Previous distance from COG "+vals.getTotalDistance());

h.setTotalDistance(vals.getTotalDistance()+h.getDeltaDistance());
getLogger().debug("Total distance is "+h.getTotalDistance()+" adding delta distance
"+h.getDeltaDistance());
getLogger().debug("");

h.setDeltavelocity(h.getCurrentVelocity() - vals.getCurrentVelocity());


String status="";

if (h.getCurrentVelocity() > 0 && h.getDeltavelocity() > 0.0) {  //positive increase
        status="SPEEDS.UP POSITIVE VELOCITY";
} else if (h.getCurrentVelocity() > 0 && h.getDeltavelocity() < 0.0) {
        status="SLOWS.DOWN POSITIVE VELOCITY";

} else if (h.getCurrentVelocity() < 0 && h.getDeltavelocity() < 0.0) { //negative increase
        status="SPEEDS.UP NEGATIVE VELOCITY";
} else if (h.getCurrentVelocity() > 0 && h.getDeltavelocity() > 0.0) {
        status="SLOWS.DOWN NEGATIVE VELOCITY";

} else {
        status="STOPPED!";
}
getLogger().debug("");

if (vals.getTotalDistance() != 0) {
```

```
                        getLogger().debug("Delta velocity is "+status+" "+h.getDeltavelocity()+"(+speeds.up -
slows.down) (new:"
                                        +h.getCurrentVelocity()+"-old:"+vals.getCurrentVelocity()+")");
                }



                getLogger().debug("---------------------------------------------------------- ITERATION END, CHECK
CRITERIAS ");

                checkTrajectoryCriterias(vals.getCriterias(), h);

                return h;
        }


        /*
         *
         * check the criterias
         *
         */
        public boolean checkTrajectoryCriterias(ArrayList<PiSpaceCriteria> criterias, PiSpaceObjectTrajectoryInfo t) {

                boolean met = false;

                for (PiSpaceCriteria p : criterias) {
                        p.perform(t);
                        if (p.isCriteriaMet()) {
                                met = true;
                                p.report(t);
                        }
                }

                return met;
        }




        /**
         * @param args
         *
         */
        public static void main(String[] args) {
```

```
	}

}
```

.....

```
	@Test
	public void testDistanceCalculationJustGoingUp() {

		//This does not test going up and then coming back down, just going up

		PiSpaceFormulas pse = new PiSpaceFormulas(
					PiSpaceFormulas.UNIT_METRES_PER_HOUR);  //I will fix up units later

		getLogger().debug("180 degrees Straight up, use Pi-Space Formulas");

		double mass1 = 100;

		double timems = 1000;

		//Planet p = new Planet(5.98E24,6.378E6,"EARTH");
		//planets.put("EARTH", p);

		double massOfEarth = 5.98E24;
		double radiusOfEarth = 6.378E7;
		double distanceFromCOG = radiusOfEarth;
		double steps = 5;
		double velocity = 100.0;
		double earthGravity = 9.809637070728721;


		//Use the trajectory

		PiSpaceObjectTrajectoryInfo t = new PiSpaceObjectTrajectoryInfo(
					0, distanceFromCOG, //x1,y1 position
					0, 0,  //x2,y2 cog
					velocity, //velocity
					180, //direction of movement
					timems, //milliseconds
					radiusOfEarth,  //radius of planet
```

```
                        mass1,  //mass of object
                        massOfEarth, //mass of earth
                        true //use Pi-Space formulas
                        );

        for (int i = 0; i < steps; i++) {
                getLogger().debug("STEP "+(i+1));
                t = pse.getNewtonianPiSpaceTrajectoryNextPosition(t);
        }

        getLogger().debug("Total distance summing trajectory is "+t.getTotalDistance());


        //Use the Newtonian approach, summing it all up in one function

        double vt = pse.getPiSpaceDistanceUseNewtonianUnits(
                        velocity, -earthGravity, steps, true);

        getLogger().debug("Total Pi-Space function distance is "+vt);

        vt = pse.getNewtonianDistance(
                        velocity, -earthGravity, steps);

        getLogger().debug("Total Newtonian function distance is "+vt);

        double accuracy = 0.1;

        assertEquals("Passed",
                        t.getTotalDistance(),
                        vt,
                        accuracy);



}
```